

A Software Tool Combining Fault Masking with User-Defined Recovery Strategies

Vincenzo De Florio, Geert Deconinck, Rudy Lauwereins
Katholieke Universiteit Leuven
Electrical Engineering Department, ACCA Group,
Kard. Mercierlaan 94, B-3001 Heverlee, Belgium.

Short title of paper: Combining Fault Masking with User-Defined Recovery

Abstract

We describe the voting farm, a tool which implements a distributed software voting mechanism for a number of parallel message passing systems. The tool, developed in the framework of EFTOS (Embedded Fault-Tolerant Supercomputing), can be used in stand-alone mode or in conjunction with other EFTOS fault tolerance tools. In the former case, we describe how the mechanism can be exploited, e.g., to implement restoring organs (N -modular redundancy systems with N -replicated voters); in the latter case, we show how it is possible for the user to implement in an easy and effective way a number of different recovery strategies via a custom, high-level language. Combining such strategies with the basic fault masking capabilities of the voting tool makes it possible to set up complex fault-tolerant systems such as, for instance, N -and- M -spare systems or gracefully degrading voting farms. We also report about the impact that our tool can have on reliability, and we show how, besides structural design goals like fault transparency, our tool achieves replication transparency, a high degree of flexibility and ease-of-use, and good performance.

1 Introduction

We herein describe the EFTOS voting farm (VF), a class of C functions implementing a distributed software voting mechanism developed in the framework of the ESPRIT-IV project 21012 EFTOS (Embedded Fault-Tolerant Supercomputing) [1, 2].

VF can be considered both as a stand-alone tool for fault masking, and as a basic block in a more complex fault tolerance structure set up within the EFTOS fault tolerance framework. Accordingly, we first draw, in Sect. 2, the design and the structure of the stand-alone voting farm, as a means for orchestrating redundant resources with fault transparency as primary goal. There, we also describe how the user can exploit the stand-alone VF tool to straightforwardly set up systems consisting of redundant modules and based on voters, e.g., restoring organs (or N -modular redundancy systems with N -replicated voters). We also report about the fault model of our tool.

In a second step (Sect. 3) we concentrate on the special extra features that VF can inherit when run as a fully EFTOS-compliant tool. To this end we describe the EFTOS fault tolerance layers and show how it is possible to make use of them in order to couple the fault masking capabilities of VF with the fault tolerance capabilities of some EFTOS tool—for instance, adding spares to an NMR system or allowing a voting farm to gracefully degrade in the presence of unrecoverable faults—and we analyse in a special case the impact that this approach can have on reliability. In the same Section we also deal with other features that can be inherited by VF from the EFTOS framework, e.g., the availability of a presentation layer for hypermedia monitoring and fault-injection.

Section 4 deals with the impact on reliability.

Section 5 deals with the analysis of the performance of VF and reports also on resource consumption and some optimisations.

Section 6 concludes this work summarizing the current state of the tool. There we also briefly describe a case study where VF has been used in developing the software stable storage device for the High Voltage Substations (HVS) of ENEL, the main Italian electricity supplier.

2 Basic structure and features of the EFTOS voting farm

A well-known approach to achieve fault masking and therefore to hide the occurrence of faults is the N -modular redundancy technique (NMR), valid both on hardware and at software level. To overcome the shortcoming of having one voter, whose failure leads to the failure of the whole system even when each and every other module is still running correctly, it is possible to use N replicas of the voter and to provide N copies of the inputs to each replica, as described in Fig.1. This approach exhibits among others the following properties:

1. Depending on the voting technique adopted in the voter, the occurrence of a limited number of faults in the inputs to the voters may be masked to the subsequent modules [3]; for instance, by using majority voting, up to $\lceil N/2 \rceil - 1$ faults can be made transparent.
2. If we consider a pipeline of such systems, then a failing voter in one stage of the

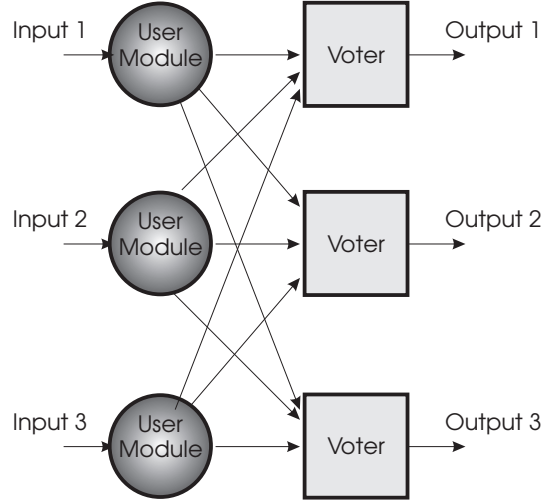


Figure 1: A restoring organ, i.e., a N -modular redundant system with N voters, when $N = 3$.

pipeline can be simply regarded as a corrupted input for the next stage, where it will be restored.

The resulting system is easily recognizable to be more robust than plain NMR, as it exhibits no single-point-of-failure. Dependability analysis confirms intuition. Property 2. in particular explains why such systems are also known as “restoring organs” [4].

From the point of view of software engineering, this system though has two major drawbacks:

- Each module in the NMR must be aware of and responsible for interacting with the whole set of voters;
- The complexity of these interactions, which is a function increasing quadratically with N , the cardinality of the voting farm, burdens each module in the NMR.

The two above observations have been recognized by us as serious impairments to our design goals, which included replication transparency, ease of use, and flexibility [5].

In order to reach the full set of our requirements, we slightly modified the design of the system as described in Fig. 2: In this new picture each module only has to interact with, and be aware of *one* voter, regardless the value of N . Moreover, the complexity of such a task is fully shifted to the voter, i.e., transparent to the user.

The basic component of our tool is therefore the *voter* (see Fig.3) which we define as follows:

A voter is a local software module connected to *one* user module and to a farm of fully interconnected fellows. Attribute “local” means that both user module and voter run on the same processing node.

As a consequence of the above definition, the user module has no other interlocutor than its voter, whose tasks are completely transparent to the user module. It is therefore

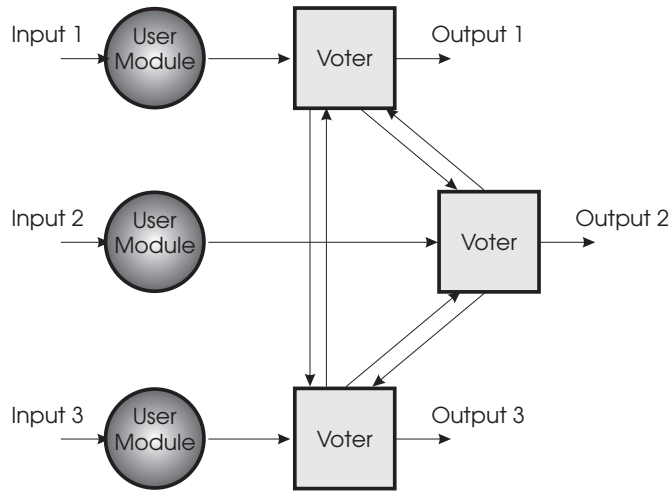


Figure 2: Structure of the EFTOS voting farm mechanism for a NMR system with $N = 3$ (the well-known triple modular redundancy system, or TMR).

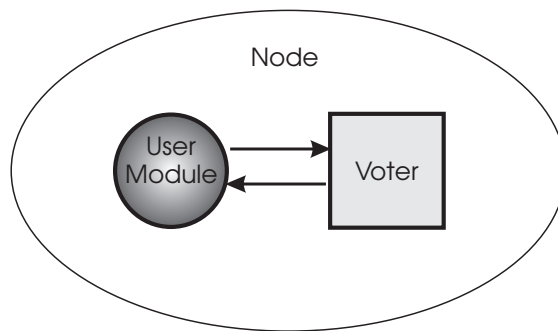


Figure 3: A user module and its voter. The latter is the only member of the farm of which the user module should be aware of: from his/her point of view, messages will only flow between these two ends. This has been designed so to minimize the burden of the user module and to keep it free to continue undisturbed as much as possible.

possible to model the whole system as a simple client-server application: on each user module the same client protocol applies (see Sect. 2.1) while the same server protocol is executed on every instance of the voter (see Sect. 2.2).

2.1 Client-Side of the Voting Farm: the User Module

Table 1 gives an example of the client-side protocol to be executed on each processing node of the system in which a user module runs: a well-defined, ordered list of actions has to take place so that the voting farm be coherently declared and defined, described, activated, controlled, and queried: In particular, *describing* a farm stands for creating a static map of the allocation of its components; *activating* a farm substantially means spawning the local voter (Sect. 2.2 will shed more light on this); *controlling* a farm means requesting its service by means of control and data messages; finally, a voting farm can also be *queried* about its state, the current voted value, etc.

As already mentioned, the above steps have to be carried out in the same way on each user module: this coherency is transparently supported in Single-Process, Multiple-Data (SPMD) architectures. This is the case, for instance, of Parsytec EPX (*Embedded Parallel eXtensions to UNIX*, see, e.g., [6, 7]) whose “initial load mechanism” transparently runs the same executable image of the user application on each processing node of the user partition.

This protocol is available to the user as a class-like collection of functions dealing with opaque objects referenced through pointers. A tight resemblance with the `FILE` set of functions of the standard C language library [8] has been sought so to shorten as much as possible the user’s learning time—the API and usage of VF closely resemble those of `FILE` (see Table 2).

VF has been crafted out using the CWEB system of structured documentation [9], which we found to be an invaluable tool both at design and at development time [10].

2.2 Server-Side of the Voting Farm: the Voter

The local voter thread represents the server-side of the voting farm. After the set up of the static description of the farm (Table 1, Step 3) in the form of an ordered list of processing node identifiers (positive integer numbers), the server-side of our application is launched by the user by means of the `VF_run()` function. This turns the static representation of a farm into an “alive” (running) object, the voter thread.

This latter connects to its user module via inter-process communication means (“local links”) and to the rest of the farm via synchronous, blocking channels (“virtual links”). We assume the availability of a means to send and to receive messages across these links—let us call these functions `Send()` and `Receive()`. Furthermore, we assume that `Send()` blocks the caller until the communication system has fully delivered the specified message to the specified (single) recipient, while `Receive()` blocks the caller until the communication system has fully transported a message directed to the caller, or until a user-specified timeout has expired.

Once the connection is established, and in the absence of faults, the voter reacts to the arrival of the user messages as a finite-state automaton: in particular, the arrival of input messages triggers a number of broadcasts among the voters—as shown in Fig.4—which are managed through the distributed algorithm described in Table 3.

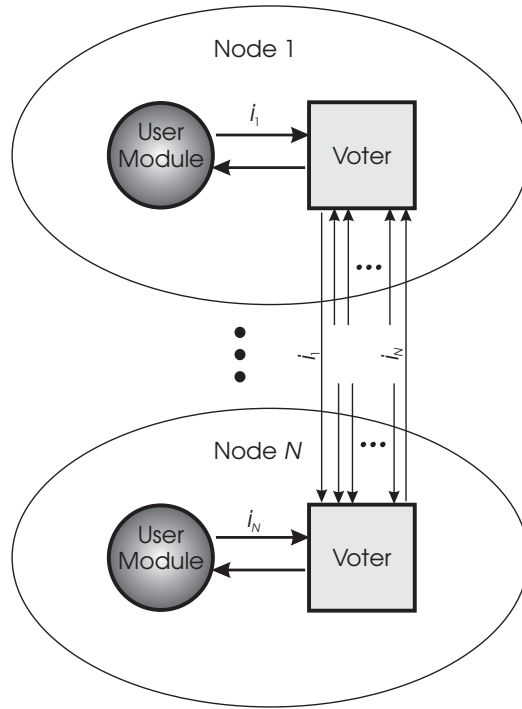


Figure 4: The “local” input value has to be broadcast to $N - 1$ fellows, and $N - 1$ “remote” input values have to be collected from each of the fellows. The voting algorithm takes place as soon as a complete set of values is available.

Assumptions of that algorithm are fail/stop behaviour and a partially synchronous system such that upper bounds are known for communication delays. This last assumption is rather realistic at least in parallel environments like EPX, which are equipped with a fast communication subsystem for their own use, so that processors do not have to compete “too much” for the network. Such subsystem also offers a reliable communication means and allows to transparently tolerate faults like, e.g., the break of a link, or a router’s failure.

When faults occur and affect up to $M < N$ voters, no arrival for more than Δt time units is interpreted as an error. As a consequence, variable `input_messages` is incremented as if a message had arrived, and its faulty state is recorded. This way we can tolerate up to $M < N$ errors at the cost of $M\Delta t$ time units. Note that even though this algorithm tolerates up to $N - 1$ faults, the voting algorithm may be intrinsically able to cope with much less than that: for instance, majority voting fails in the presence of faults affecting $\lceil N/2 \rceil$ or more voters. As another example, algorithms computing a weighted average of the input values consider all items whose “faulty bit” is set as zero-weight values, automatically discarding them from the average. This of course may also lead to imprecise results as the number of faults gets larger.

Besides the input value, which represents a request for voting, the user module may send to its voter a number of other requests—some of these are used in Table 1, Step 5. In particular, the user can choose to adopt a voting algorithm among the following ones:

- Formalized majority voting technique,
- Generalized median voting technique,
- Formalized plurality voting technique,
- Weighted averaging technique,
- Consensus,

the first four items being the voting techniques that were generalized in [3] to “arbitrary N -version systems with arbitrary output types using a metric space framework.” To use these algorithms, a metric function can be supplied by the user when he/she “opens” the farm (Table 1, Step 2, function `objcmp()`): this is exactly the same approach used in opaque C functions like e.g., `bsearch()` or `qsort()` [8]. A default metric function is also available.

The choice of the algorithm, as well as other control choices are managed via function `VF_control()`, which takes as argument a voting farm pointer plus a variable number of control argument—in Table 1, Step 5, these arguments are an input message, a virtual link for the output vote, an algorithm identifier, plus an argument for that algorithm.

Other requests include the setting of some algorithmic parameters and the removal of the voting farm (function `VF_close()`).

The voters’ replies to the incoming requests are straightforward. In particular, a `VF_DONE` message is sent to the user module when a broadcast has been performed; for the sake of avoiding deadlocks, one can only close a farm after the `VF_DONE` message has been sent. Any failed attempt causes the voter to send a `VF_REFUSED`

message. The same refusing message is sent when the user tries to initiate a new voting session sooner than the conclusion of the previous session.

Note how function `VF_get()` (Table 1, Step 6) simply sets the caller in a waiting state from which it exits either on a message arrival or on the expiration of a time-out.

2.3 Fault Model

VF can deal with the following classes of faults [11]:

- physical as well as human-made,
- accidental as well as intentional,
- development as well as operational,
- internal and external faults,
- permanent and temporary,

as long as the corresponding failure domain consists only of value failures. Timing errors are also considered, though the delay must not be larger than some bounded value. The tool is only capable of dealing with one fault at a time—the tool is ready to deal with other new faults only after having recovered from the present one. Consistent value errors are tolerated. Under this assumption, arbitrary in-code value errors may occur (the adopted metric approach is not able to deal with non-code values).

3 The voting farm as a fully EFTOS-compliant tool

The above section describes VF as a stand-alone tool. One of the consequences of this strategy is the adoption of fault masking as a means to make some faults transparent, e.g., via NMR-systems. In this Section we now shift the attention to a more general approach towards tolerating faults which has been defined as the object of the ESPRIT project EFTOS. We show in particular how the conjoint use of VF and of the EFTOS framework may yield systems that combine more than one fault tolerance technique, and as such are more dependable.

To this end, we first introduce EFTOS and its framework, then we describe how to make use of its extra features. Section 4 investigates on the impact that this version of VF can have on reliability in a special case.

3.1 EFTOS and its Framework

The overall object of the ESPRIT-IV Project 21012 EFTOS [1, 2] (Embedded Fault-Tolerant Supercomputing) has been to set up a software framework for integrating fault tolerance into embedded distributed high-performance applications in a flexible, effective, and straightforward way. The EFTOS framework has been first implemented on a Parsytec CC system [7], a distributed-memory MIMD supercomputer consisting of processing nodes based on PowerPC 604 microprocessors at 133MHz, dedicated high-speed links, I/O modules, and routers. As part of the Project, this framework has been then ported to a Microsoft Windows NT / Intel PentiumPro platform and to a TEX

/ DEC Alpha platform [12, 13] so to fulfill the requirements of the EFTOS application partners. We herein constantly refer to the version running on the CC system and its operating system, a UNIX-dialect called EPX/nK (Parsytec's nano-kernel version of EPX.)

The main characteristics of the CC system are the adoption of the thread processing model and of the message passing communication model: communicating threads exchange messages through a proprietary message passing library called EPX [6]. The hypotheses on the target communication system we drew in Sect. 2.2 follow closely the peculiar characteristics of EPX. In particular, EPX adopts the concepts of local and virtual links, and its `Send()` and `Receive()` are compliant with what is required in Sect. 2.2.

Through the adoption of the EFTOS framework, the target embedded parallel application is plugged into a hierarchical, layered system (see Fig. 5) whose structure and basic components are:

- At the base level, a distributed net of “servers” whose main task is mimicking possibly missing (with respect to the POSIX standards) operating system functionalities, such as remote thread creation;
- One level upward (detection tool layer), a set of parametrisable functions managing error detection (we call them “Dtools”). These basic components are plugged into the embedded application to make it more dependable. EFTOS supplies a number of these Dtools, e.g., a watchdog timer thread and a trap-handling mechanism, plus an API for incorporating user-defined EFTOS-compliant tools;
- At the third level (control layer), a distributed application called *DIR net* (detection, isolation, and recovery network) [14] is available to coherently combine the Dtools, to ensure consistent strategies throughout the whole system, and to play the role of a backbone handling information to and from the fault tolerance elements.

The DIR net can also be regarded as a fault-tolerant network of crash-failure detectors, connected to other peripheral error detectors. It can be used as a general toolset in which fundamental distributed algorithms like, e.g., those drawn in [15], can be implemented;

- At the fourth level (application layer), the Dtools and the components of the DIR net are combined into dependable mechanisms i.e., methods to guarantee fault-tolerant communication, stable storage devices [17], the voting farm mechanism, etc;
- The highest level (presentation layer) is given by a hypermedia distributed application based on standard World-Wide Web technology and on Tcl/Tk [18], which renders the structure and the state of the user application [19].

An important feature of the above sketched layered system is the ability to execute user-defined recovery strategies:

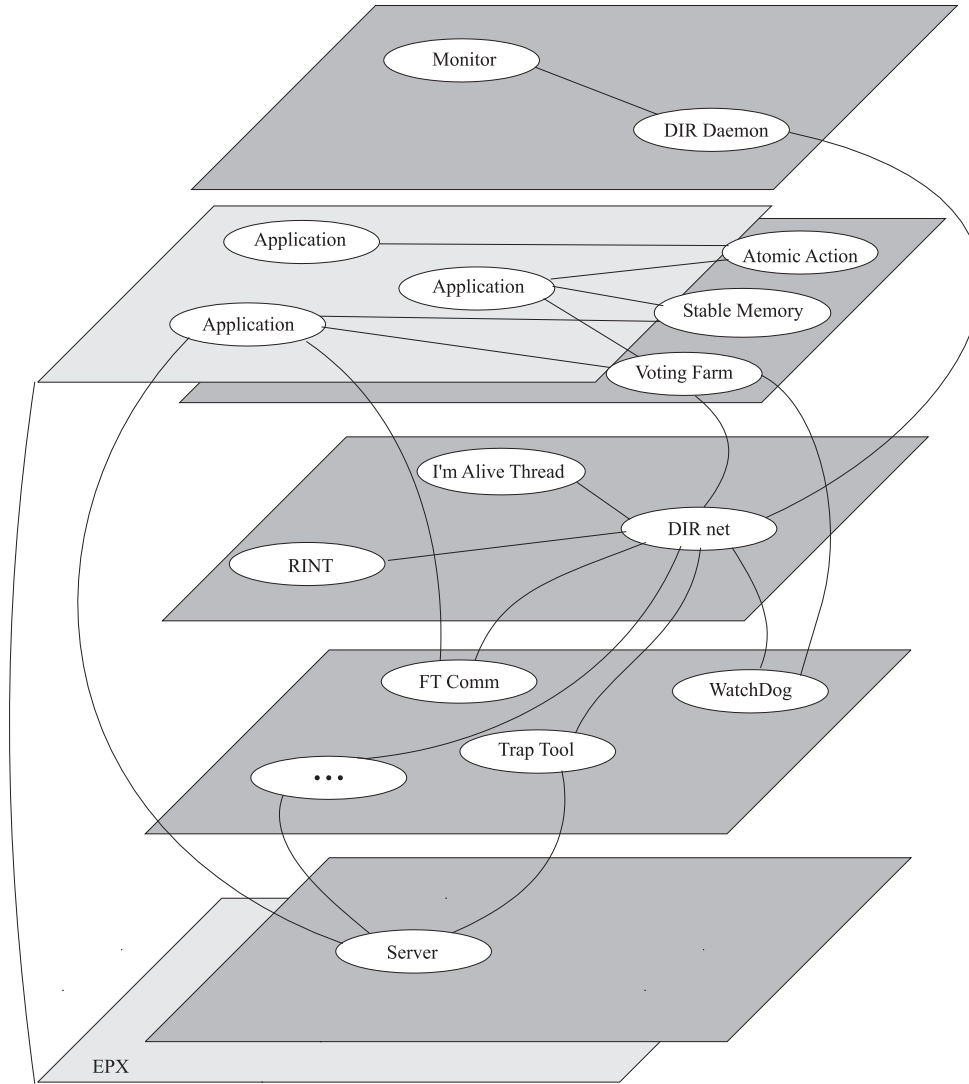


Figure 5: The structure of the EFTOS framework. Light gray has been used for the operating system and the user application, while dark gray layers pertain EFTOS.

3.2 The EFTOS Recovery Language

Apart from its main role of backbone of the EFTOS fault tolerance framework, the DIR net also coordinates a virtual machine, called the recovery interpreter (RINT for short), meant to execute user-defined recovery strategies which are available at this level as a set of recovery opcodes (r-codes for short) [16].

A special high-level language has been set up, called Recovery Language (RL), by means of which the user can compose custom-made recovery strategies. Strategies are then translated into the lower level r-codes by means of a translator.

A RL specification is a collection of isolation, reconfiguration, and recovery actions guarded by selective conditions (IF's). Recovery actions include rebooting or shutting down a node, killing or restarting a thread or a logical group of threads, sending warnings to single or grouped threads, and functions to purge error-records from the database of the DIR net.

The scheme works as follows: the user writes with RL a recovery strategy, viz. a specification of some actions to be taken to tackle each particular error condition as that error is detected. For each error, the DIR net awakes the RINT thread, which interprets the r-code equivalent of the RL source, looking for fulfilled conditions, possibly accessing the system database maintained by the DIR net. Once a condition is met, the corresponding actions are executed, which are supposed to be able to tackle the error. A default action is also available in case no IF is evaluated as true. The whole strategy is depicted in Fig. 6.

During the lifetime of the application, this framework guards it from a series of possible deviations from the expected activity; this is done either by executing detection, isolation, and reconfiguration tasks, or by means of fault masking—this latter being provided by the EFTOS voting farm, which we are going to describe in the Section to follow.

As a last remark, the EFTOS framework appears to the user as a library of functions written in the C programming language.

3.3 Using the Voting Farm in Conjunction with the EFTOS Framework

What we described in Sect. 2 might be referred to as “the Voting Farm in stand-alone mode”, i.e., unplugged from its originating environment, the EFTOS framework. Here we describe further capabilities and features that are offered to the user of our tool when run as part of a fully EFTOS-compliant application.

When used in conjunction with the EFTOS DIR net, each voter is transparently connected to a DIR net component and to a watch-dog timer Dtool, connected in turn to the DIR net. This way the DIR net is informed of errors affecting the voters. Each voter also directly informs the DIR net of its state transitions (we call them “phases”) by sending phase-identifiers (pids) that can assume one of the following values:

VFP_INIT, that is, the voter is waiting for the first input value from the user,

VFP_BROADCAST, meaning the voter is currently broadcasting the input value to its fellows,

VFP_VOTING, i.e., the voter has entered the voting function,

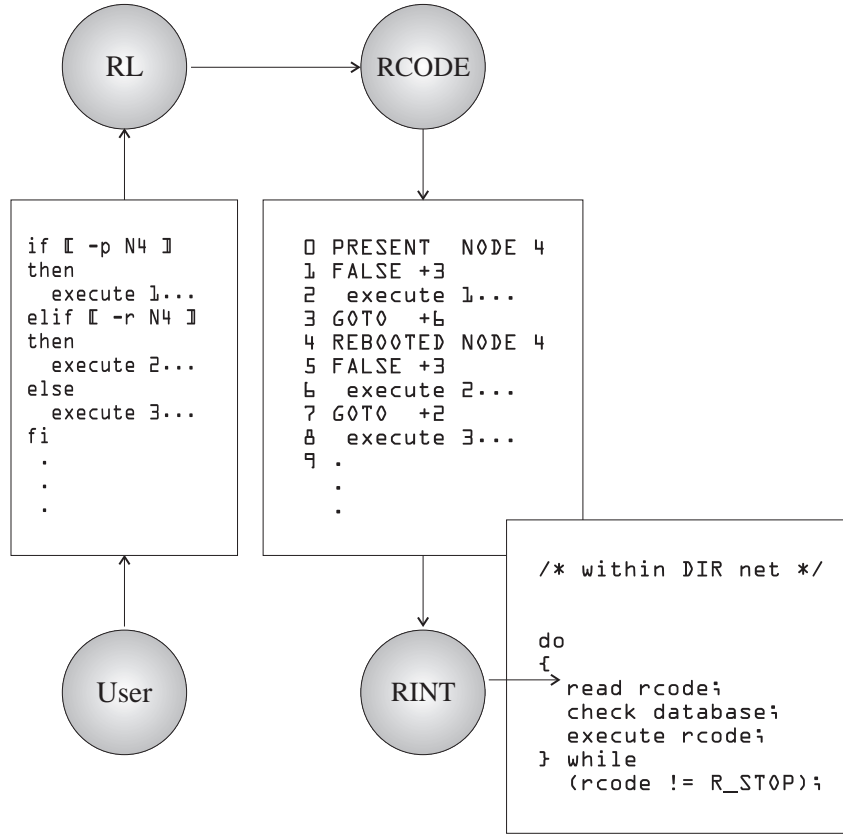


Figure 6: A global view of an RL program: the user supplies an RL source code; the rl translator turns it into binary r-codes; the r-codes are interpreted at run time by the RINT thread within the DIR net.

VFP_SUCCESS, i.e., an output vote has been produced and the voter is back in its waiting state;

VFP_FAILURE, which means the voting function was not able to produce an output vote.

The pid is then stored by the DIR net in its global database [14] and can be used during the interpretation of the r-codes (see Sect. 3.2).

With RL the user can express strategies aiming for instance at substituting a suspected voter with a non-faulty deputy run elsewhere in the system (see Table 4), or at a graceful degradation of the system (e.g., by killing those voters which are in phase VFP_FAILURE; see Table 5). Other strategies are up to the user.

Another feature that can be inherited by VF when used as part of the EFTOS framework comes from its presentation layer: the farm can be monitored via the EFTOS visualization tool [19], a hypermedia distributed application which remotely pilots a World-Wide Web browser so to render the structure and the outcome of the voting sessions.

4 Impact on Reliability

Reliability can be greatly improved by this technique. For instance, using Markov models, under the assumption of independence between faults occurrence, it is possible to show that, let $R(t)$ be the reliability of a single, non-replicated component, then

$$R^{(0)}(t) = 3R(t)^2 - 2R(t)^3, \quad (1)$$

i.e., the equation expressing the reliability of a TMR system, can be considerably improved by adding one spare, even in the case of non-perfect error detection coverage. This is the equation resulting from the Markov model in Fig. 7, expressed as a function of error recovery coverage (C , defined as the probability associated with the process of identifying the failed module out of those available and being able to switch in the spare [4]) and time (t):

$$R^{(1)}(C, t) = (-3C^2 + 6C) \times [R(t)(1 - R(t))]^2 + R^{(0)}(t). \quad (2)$$

Appendix A gives some mathematical details on Eq. (2).

Adding more spares obviously implies further improving reliability. In general we can think of a class of monotonically increasing reliability functions,

$$(R^{(M)}(C, t))_{M \geq 0}, \quad (3)$$

corresponding to systems adopting $N + M$ replicas. Depending on both cost and reliability requirements, the user can choose the most-suited values for M and N .

Fig. 8 compares Eq. (1) and (2) in the general case (left picture) and under perfect coverage (right picture). In this latter case, the reliability of a single, non-redundant system is also portrayed. Note furthermore how the crosspoint between the three-and-one-spare system and the non-redundant system is considerably lower than the crosspoint between this latter and the TMR system— $R(t) \approx 0.2324$ vs. $R(t) = 0.5$. The reliability of the system can therefore be increased from the one of a pure NMR system to that of N -and- M -spare systems (see Fig. 8).

5 Performance of the Voting Farm

This Section first focuses on estimating the voting delay via some experiments. Then it summarises the overheads in terms of system resources (threads, memory, links...) Finally it briefly sketches a formal model used to evaluate the broadcast algorithm in Table 3 and reports on some analytical results out of it.

5.1 Time and Resources Overheads of the Voting Farm.

All measurements have been performed running a restoring organ consisting of N processing nodes, $N = 1, \dots, 4$. The executable file has been obtained on a Parsytec CC system with the `ancc` C compiler using the `-O` optimization flag. During the trials the CC system was fully dedicated to the execution of that application.

The application has been executed in four runs, each of which has been repeated fifty times, increasing the number of voters from 1 to 4, one voter per node. Wall-clock times have been collected. Averages and standard deviations are shown in Table 6.

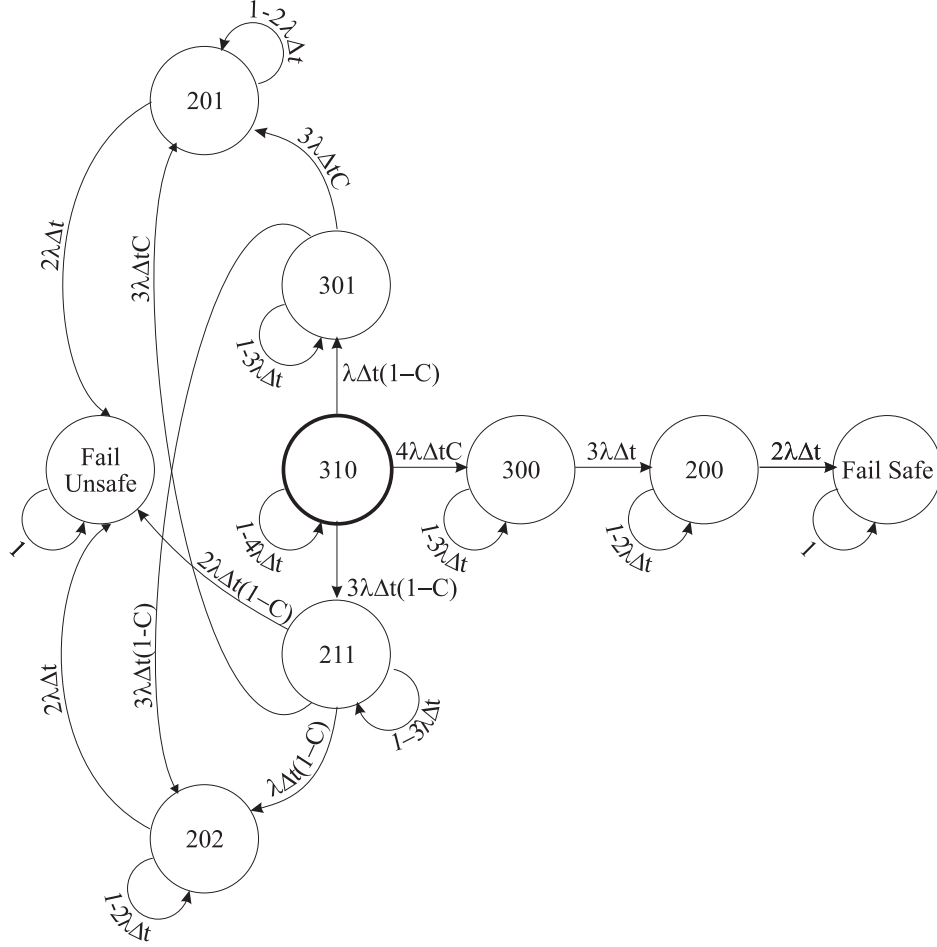


Figure 7: Markov reliability model for a TMR-and-1-spare system. λ is the failure rate, C is the error recovery coverage factor. ‘Fail safe’ state is reached when the system is no more able to correctly perform its function, though the problem has been safely detected and handled properly. In ‘Fail unsafe,’ on the contrary, the system is incorrect, though the problem has not been handled or detected. Every other state is labeled with three digits, $d_1d_2d_3$, such that d_1 is the number of non-faulty modules in the TMR system, d_2 is the number of non-faulty spares (in this case, 0 or 1), and d_3 is the number of undetected, faulty modules. The initial state, 310, has been highlighted. This model is solved by Eq. (2).

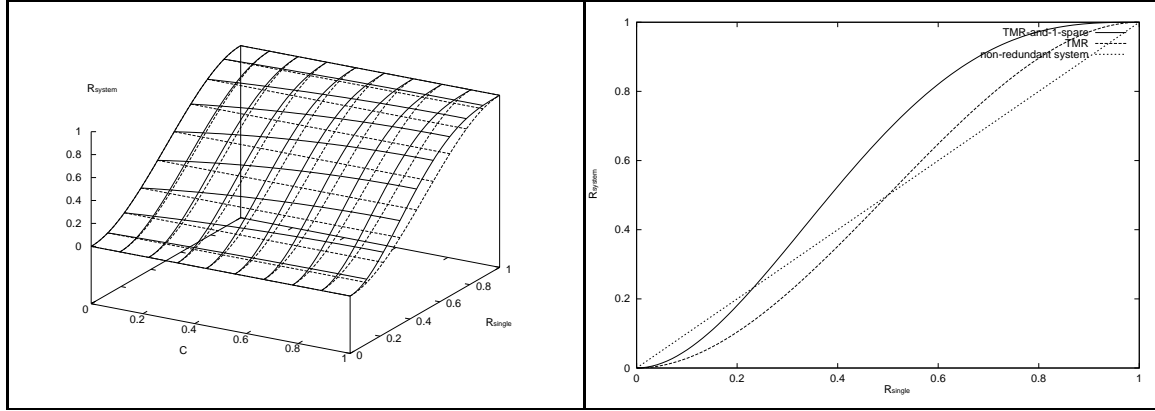


Figure 8: Graphs of Eq. (1) and (2). This latter is strictly above the former in both pictures. The right picture considers the special case of $C = 1$ (perfect error detection coverage) and draws also the reliability of a single, non-redundant system.

As of the overhead in resources, N threads have to be spawned, and N local links are needed for the communication between each user module and its local voter. The network of voters calls for another $N \times (N - 1)/2$ virtual links.

5.2 Optimizations

Overall performance is obviously conditioned, among other factors, by both the nature of the broadcast algorithm as well as by the diameter of the communication network. In particular we formally proved [20] that, in a fully synchronized, fully connected (crossbar) system, the execution time of the algorithm we adopted can vary, depending on the permutation of the sequence of **Send()**'s which constitute the broadcast, from $O(N^2)$ (identity permutation) to $O(N)$ (one-cycled permutation). In the case of the one-cycled permutation, we showed that the efficiency of the channel system is constant with respect to N , and approximately equal to 66.67%.

Due to the above results, we consider as part of the process of porting VF to another platform an optimization step in which a best-performing sequence is selected. The user is also allowed to substitute altogether the default broadcast function with another one.

6 Conclusions

A flexible, easy to use, efficient mechanism for software voting has been described. In particular, it has been shown how it is possible to combine fault masking with recovery techniques when the mechanism is coupled with other tools of the EFTOS framework. In particular, the availability of a custom, high-level language for expressing recovery strategies allows to decouple aspects related to fault tolerance programming from those related to a system's normal operation, with a profitable trade-off between the need for transparent fault tolerance and the need to orchestrate software fault tolerance in the application layer (see e.g., [21, 22].)

The tool is currently available for a number of message passing environments, including Parsytec EPX, Windows/NT, and TXT TEX. A special, “static” version has been developed for this latter, which adopts the mailbox paradigm as opposed to message passing via virtual links. In this latter version, the tool has been used in a software fault tolerance implementation of a stable memory system (SMS) for the high-voltage substation controller of ENEL, the main Italian electricity supplier [17]. This SMS is based on a combination of temporal and spatial redundancy to tolerate both transient and permanent faults, and uses two voting farms, one with consensus and the other with majority voting. The tool proved to fulfill its goals and will gradually substitute the dedicated hardware board originally implementing stable storage at ENEL.

Acknowledgments. This project is partly supported by an FWO Krediet aan Navorsers, by the ESPRIT-IV projects 21012 “EFTOS” and 28620 “TIRAN,” and by COF/96/11. Vincenzo De Florio is on leave from the Tecnopolis CSATA Novus Ortus science park. Geert Deconinck is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (FWO). Rudy Lauwereins is a Senior Research Associate of FWO.

References

- [1] DECONINCK, G., DE FLORIO, V., LAUWEREINS, R., AND VARVARIGOU, T.: ‘EFTOS: A software framework for more dependable embedded HPC applications,’ *Proc. of the 3rd Int. Euro-Par Conference, EuroPar’97, Lecture Notes in Computer Science*, 1997, **1300**, pp. 1363–1368
- [2] DECONINCK, G., VARVARIGOU, T., BOTTI, O., DE FLORIO, V., KONTIZAS, A., TRUYENS, M., ROSSEEL, W., LAUWEREINS, R., CASSINARI, F., GRAEBER, S., AND KNAACK, U.: ‘(Reusable software solutions for more fault-tolerant) Industrial embedded HPC applications,’ *Supercomputer XIII*, 1997, **69** pp. 23–44
- [3] LORCZAK, P. R., CAGLAYAN, A. K., AND ECKHARDT, D. E.: ‘A theoretical investigation of generalized voters for redundant systems,’ *Proc. of the 19th Int. Symposium on Fault-Tolerant Computing, FTCS-19*, June 1989, Chicago, IL, pp. 444–451
- [4] JOHNSON, B. W.: ‘*Design and Analysis of Fault-Tolerant Digital Systems*’ (Addison-Wesley, New York, 1989)
- [5] DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R.: ‘The EFTOS voting farm: a software tool for fault masking in message passing parallel environments,’ *Proc. of the 24th EuroMicro Conf. on Engineering Systems and Software for the Next Decade – Workshop on Dependable Computing Systems*, Euromicro’98, August 1998, Västerås, Sweden
- [6] ANONYMOUS: ‘*Embedded Parix Programmer’s Guide*,’ Parsytec CC Series Hardware Documentation (Parsytec GmbH, Aachen, Germany, 1996)
- [7] ANONYMOUS: ‘*Parsytec CC Series—Cognitive Computing*’ (Parsytec GmbH, Aachen, Germany, 1996)
- [8] KERNIGHAN, B. W., AND RITCHIE, D. M.: ‘*The C Programming Language*’ (Prentice-Hall, Englewood Cliffs, NJ, 1988) 2nd edn.
- [9] DE FLORIO, V.: ‘The voting farm—a distributed class for software voting,’ Tech. Rep., ESAT/ACCA/1997/3, Katholieke Universiteit Leuven, June 1997
- [10] KNUTH, D. E.: ‘Literate programming,’ *The Comp. Jour.*, 1984, **27** pp. 97–111
- [11] LAPRIE, J. C.: ‘Dependability—its attributes, impairments and means,’ in RANDELL, B., LAPRIE, J. C., KOPETZ, H., AND LITTLEWOOD, B. (editors): ‘*ESPRIT Basic Research Series: Predictably Dependable Computing Systems*’ (Springer-Verlag, Berlin, 1995) pp. 3–18.
- [12] ANONYMOUS: ‘*TEX User Manual*’ (TXT Ingegneria Informatica, Milano, Italy, 1997)
- [13] ANONYMOUS: ‘*CS_Q66E Alpha Q-Bus CPU module: User’s Manual*’ (Digital Equipment Corp., 1997)

- [14] TRUYENS, M., DECONINCK, G., DE FLORIO, V., ROSSEEL, W., LAUWEREINS, R., AND BELMANS, R.: 'A framework backbone for software fault-tolerance in embedded parallel applications,' accepted at the 7th EuroMicro Workshop on Parallel and Distributed Processing, PDP'99, to be held in February 1999
- [15] CHANDRA, T., AND TUEG, S.: 'Unreliable failure detectors for reliable distributed systems,' *Journal of the ACM*, 1996, **34**(1) pp. 225–267
- [16] DE FLORIO, V.: 'The EFTOS recovery language,' Tech. Rep., ESAT/ACCA/1997/4, Katholieke Universiteit Leuven, December 1997
- [17] DECONINCK, G., BOTTI, O., CASSINARI, F., DE FLORIO, V., AND LAUWEREINS, R.: 'Stable memory in substation automation: a case study,' *Proc. of the 28th Int. Symposium on Fault-Tolerant Computing*, FTCS-28, June 1998, Munich, Germany, pp. 452–457
- [18] OUSTERHOUT, J. K.: '*Tcl and the Tk Toolkit*' (Addison-Wesley, Reading, MA, 1994)
- [19] DE FLORIO, V., DECONINCK, G., TRUYENS, M., ROSSEEL, W., AND LAUWEREINS, R.: 'A hypermedia distributed application for monitoring and fault-injection in embedded parallel programs,' *Proc. of the 6th EuroMicro Workshop on Parallel and Distributed Processing*, PDP'98, January 1998, Madrid, Spain, pp. 349–355
- [20] DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R.: 'The algorithm of pipelined broadcast,' submitted to *Information and Computation*, March 1998
- [21] HUANG, Y., AND KINTALA, C. M. R.: 'Software Implemented Fault Tolerance: Technologies and Experience,' *Proc. of 23rd Int. Symposium on Fault-Tolerant Computing*, FTCS-23, June 1993, Toulouse, France. Also appeared as a chapter in LYU, M., Editor: '*Software Fault Tolerance*' (John Wiley & Sons, New York, 1995)
- [22] SALTZER, J. H., REED, D. P., AND CLARK, D. D.: 'End-to-end arguments in system design,' *ACM Transactions on Computer Systems*, 1984, **2**(4) pp. 277–288

A Appendix: Mathematical Details

Here we describe the basic steps leading to Eq. 2. The Markov reliability model of Fig. 7 brings to the following set of equations:

$$\left\{ \begin{array}{l} p_{310}(t + \Delta t) = p_{310}(t)(1 - 4\lambda\Delta t) \\ p_{300}(t + \Delta t) = p_{300}(t)(1 - 3\lambda\Delta t) + p_{310}(t)4\lambda\Delta tC \\ p_{200}(t + \Delta t) = p_{200}(t)(1 - 2\lambda\Delta t) + p_{300}(t)3\lambda\Delta t \\ p_{\text{FS}}(t + \Delta t) = p_{\text{FS}}(t) + p_{200}(t)2\lambda\Delta t \\ p_{211}(t + \Delta t) = p_{211}(t)(1 - 3\lambda\Delta t) + p_{310}(t)3\lambda\Delta t(1 - C) \\ p_{301}(t + \Delta t) = p_{301}(t)(1 - 3\lambda\Delta t) + p_{310}(t)\lambda\Delta t(1 - C) \\ p_{201}(t + \Delta t) = p_{201}(t)(1 - 2\lambda\Delta t) + p_{301}(t)3\lambda\Delta tC + \\ \quad p_{211}(t)3\lambda\Delta tC \\ p_{202}(t + \Delta t) = p_{202}(t)(1 - 2\lambda\Delta t) + p_{301}(t)3\lambda\Delta t(1 - C) + \\ \quad p_{211}(t)\lambda\Delta t(1 - C) \\ p_{\text{FU}}(t + \Delta t) = p_{\text{FU}}(t) + p_{201}(t)2\lambda\Delta t + \\ \quad p_{211}(t)2\lambda\Delta t(1 - C) + p_{202}(t)2\lambda\Delta t \end{array} \right. \quad (4)$$

For any state s , let us now call $L_s = L(p_s(t))$, where L is the Laplace transform. Furthermore, assuming (310) as the initial state we set $p_{310}(0) = 1$ and $\forall s \neq (310) : p_s(0) = 0$. Then taking the limit of the above equations as t goes to zero and taking the Laplace transform we get to

$$\left\{ \begin{array}{l} L_{310} = \frac{1}{s+4\lambda} \\ L_{300} = \frac{4C}{s+3\lambda} - \frac{4C}{s+4\lambda} \\ L_{200} = \frac{6C}{s+4\lambda} - \frac{12C}{s+3\lambda} + \frac{6C}{s+2\lambda} \\ L_{\text{FS}} = \frac{C}{s} - \frac{3C}{s+4\lambda} + \frac{8C}{s+3\lambda} - \frac{6C}{s+2\lambda} \\ L_{211} = \frac{3(1-C)}{s+3\lambda} - \frac{3(1-C)}{s+4\lambda} \\ L_{301} = \frac{1-C}{s+3\lambda} - \frac{1-C}{s+4\lambda} \\ L_{201} = 6C(1-C)\left(\frac{1}{s+4\lambda} - \frac{2}{s+3\lambda} + \frac{1}{s+2\lambda}\right) \\ L_{202} = 3(1-C)^2\left(\frac{1}{s+4\lambda} - \frac{2}{s+3\lambda} + \frac{1}{s+2\lambda}\right) \end{array} \right. \quad (5)$$

Inverting the Laplace transform we now get to

$$\left\{ \begin{array}{l} p_{310}(t) = e^{-4\lambda t} \\ p_{300}(t) = 4Ce^{-3\lambda t} - 4Ce^{-4\lambda t} \\ p_{200}(t) = 6Ce^{-4\lambda t} - 12Ce^{-3\lambda t} + 6Ce^{-2\lambda t} \\ p_{211}(t) = 3(1-C)e^{-3\lambda t} - 3(1-C)e^{-4\lambda t} \\ p_{301}(t) = (1-C)e^{-3\lambda t} - (1-C)e^{-4\lambda t} \\ p_{201}(t) = 6C(1-C)(e^{-4\lambda t} - 2e^{-3\lambda t} + e^{-2\lambda t}) \\ p_{202}(t) = 3(1-C)^2(e^{-4\lambda t} - 2e^{-3\lambda t} + e^{-2\lambda t}) \end{array} \right. \quad (6)$$

(Only useful states have been computed.)

Let us call $R = e^{-\lambda t}$ the reliability of the basic component of the system, and R_{TMR} the reliability of the TMR system based on the same component. The reliability of the three and one spare system, $R^{(1)}(C, t)$, is given by the sum of the above probabilities:

$$\begin{aligned}
R_{(1)}(C, t) &= R^4(-3C^2 + 6C) + R^3(6C^2 - 12C - 2) + R^2(-3C^2 + 6C + 3) \\
&= (-3C^2 + 6C)(R(1 - R))^2 + (3R^2 - 2R^3) \\
&= (-3C^2 + 6C)(R(1 - R))^2 + R_{\text{TMR}},
\end{aligned}$$

which proves Eq. (2).

```

1  /* declaration */
   VotingFarm_t *vf;
2  /* definition */
   vf ← VF_open(objcmp);
3  /* description */
    $\forall i \in \{1, \dots, N\} : \text{VF\_add}(\text{vf}, \text{node}_i, \text{ident}_i);$ 
4  /* activation */
   VF_run(vf);
5  /* control */
   VF_control(vf, VF_input(obj, sizeof(VFobj_t)),
              VF_output(link),
              VF_algorithm (VFA_WEIGHTED_AVERAGE),
              VF_scaling_factor(1.0) );
6  /* query */
   do {} while (VF_error==VF_NONE   $\wedge$   VF_get(vf)==VF_REFUSED);
7  /* deactivation */
   VF_close(vf);

```

Table 1: An example of usage of the voting farm.

phase	FILE class	VotingFarm.t class
declaration	FILE* f;	VotingFarm.t* vf;
opening	f = fopen(...);	vf = VF_open(...);
control	fwrite(f, ...);	VF_control(vf, ...);
closings	fclose(f);	VF_close(vf);

Table 2: The C language standard class for managing file is compared with the VF class. The tight resemblance has been sought in order to shorten as much as possible the user's learning time.

```

1 /* each voter gets a unique voter_id  $\in \{1, \dots, N\}$  */
   voter_id  $\leftarrow$  who-am-i();
2 /* all messages are first supposed to be valid */
    $\forall i : \text{valid}_i \leftarrow \text{TRUE};$ 
3 /* keep track of the number of received input messages */
    $i \leftarrow \text{input\_messages} \leftarrow 0;$ 
4 do {
5     /* wait for an incoming message or a timeout */
       Wait_Msg_With_Timeout( $\Delta t$ );
6     /*  $u$  points to the user module's input */
       if ( Sender()  $\equiv$  USER )  $u \leftarrow i;$ 
7     /* read it */
       if (  $\neg$  Timeout )  $\text{msg}_i \leftarrow \text{Receive}();$ 
8     /* or invalidate its entry */
       else  $\text{valid}_i \leftarrow \text{FALSE};$ 
9     /* count it */
        $i \leftarrow \text{input\_messages} \leftarrow \text{input\_messages} + 1;$ 
10    if (voter_id  $\equiv$  input_messages) Broadcast( $\text{msg}_u$ );
11 } while (input_messages  $\neq$  N);

```

Table 3: The distributed algorithm needed to regulate the right to broadcast among the N voters. Each voter waits for a message for a time which is at most Δt , then it assumes a fault affected either a user module or its voter. Function **Broadcast()** sends its argument to all voters whose id is different from **voter_id**. It is managed via a special sending thread so to circumvent the case of a possibly deadlock-prone **Send()**.

```
INCLUDE "vf_phases.h"
IF [ -FAULTY THREAD1
    OR -PHASE THREAD1 == {VFP_FAILURE} ]
THEN
    KILL THREAD1
    START THREAD4 AND
    WARN THREAD2, THREAD3
FI
```

Table 4: A recovery rule coded in RL. We suppose a voting farm consisting of three threads, identified by integers 1–3. If the first thread is detected as faulty or its state is `VFP_FAILURE`, then that thread is killed, a new thread is started and the fellows of the faulty one are alerted so that they restore a non-faulty farm. Three of such rules may be used to set up, e.g., a three-and-one-spare system. (Note the `INCLUDE` statement, which is used to import C-style definitions into RL. Note also the curly brackets operator, which de-references such definitions, as in `{VFP_FAILURE}`.)

```
INCLUDE "vf_phases.h"
IF [ -FAULTY GROUP1
    OR -PHASE GROUP1 == {VFP_FAILURE} ]
THEN
    KILL THREAD@ AND WARN THREAD~
FI
```

Table 5: Another recovery rule coded in RL. We suppose a voting farm consisting of three threads, collectively identified as “group 1.” The IF statement checks whether any element of the group has been detected as faulty or is currently in VFP_FAILURE state. If so, in the first action, those that fulfill the condition (identified in RL as THREAD@) are killed, while those that do not fulfill the condition (in RL, THREAD~) are warned. This allows a graceful degradation of the voting farm.

number of nodes	average	standard deviation
1	0.000615	0.000006
2	0.001684	0.000022
3	0.002224	0.000035
4	0.003502	0.000144

Table 6: Time overhead of the voting farm for one to four node systems (one voter per node). The unit is seconds.